Modular Verification of SRT Division *

To appear in CAV '96

H. Rueß, N. Shankar, and M.K. Srivas

Computer Science Laboratory, SRI International
Menlo Park, CA 94025
{`ruess,shankar,srivas`}`@csl.sri.com`

Abstract. We describe a formal specification and verification in PVS for the general theory of SRT division, and for the hardware design of a specific implementation. The specification demonstrates how attributes of the PVS language (in particular, predicate subtypes) allow the general theory to be developed in a readable manner that is similar to textbook presentations, while the PVS `table` construct allows direct specification of the implementation's quotient look-up table. Verification of the derivations in the SRT theory and for the data path and look-up table of the implementation are highly automated and performed for arbitrary, but finite precision; in addition, the theory is verified for general radix, while the implementation is specialized to radix 4. The effectiveness of the automation derives from PVS's tight integration of rewriting with decision procedures for equality, linear arithmetic over integers and rationals, and propositional logic. This example demonstrates that the resources of an expressive specification language and of a general-purpose theorem prover are not inimical to highly automated verification in this domain, and can contribute to clarity, generality, and reuse.

1 Introduction

The SRT division algorithm is one of the most popular methods for implementing floating-point division and related operations in high-performance arithmetic units. Even though the theory of SRT division has been extensively studied [Atk68], the design of dividers still remains a serious challenge [OF94], and it is easy to make mistakes in its implementation—as was illustrated by the much publicized FDIV error in the Intel Pentium chip. As Pratt [Pra95] points in his analysis, it is unlikely testing alone would have caught that error as it was due to five wrong entries in the quotient look-up table in a region of the table that was thought to be unreachable. Hence, formal verification can play an essential role in the design and debugging of arithmetic circuits.

In this paper, we present a mechanized verification of a general SRT division algorithm that can be used for performing floating-point divisions and an implementation of it based on the circuit given in [Tay81]. This circuit implements the

* Supported in part by ARPA under Arpa Order A721, by NASA under contract NAS1-20334, and by NSF Grant No. CCR-930044. We gratefully acknowledge the valuable guidance and help given by John Rushby, Sam Owre, Ed Clarke, and Steve German.

IEEE floating-point standard, and its kernel consists of a fixed-point iteration. The verification of this kernel was performed in the interactive theorem proving system PVS [ORSvH95]. Since our goal was to perform the verification so that as much of the initial set-up effort can be reused in the verification of other similar circuits, we took a modular approach that separates concerns about general facts of the SRT theory from a specific circuit implementation and a look-up table. Furthermore, we develop clear interfaces between these parts, so that each of the verifications can be done separately.

More precisely, the formalization and verification of the SRT divider proceeds in two steps. First, we formalize textbook knowledge about SRT dividers at an algorithmic level and verify its correctness. The formalization at this level does not use a specific data path to compute the partial remainder nor a specific look-up table. It characterizes a set of semantic constraints a look-up table ought to satisfy and a recurrence relation a partial remainder computation circuit ought to preserve. In the second step, we specify a data path circuit (bit-vector signals over time) to compute the partial remainder and define a specific look-up table, both of which are based on the implementation given in [Tay81]. We then show that the data path circuit and the look-up table meet the constraints characterized in step one. Both steps of the verification are performed for arbitrary, but finite precision, which appears as a parameter to the specification. The first step of the verification is applicable to arbitrary radixes, while the second step assumes a radix-4 implementation, since it uses a look-up table for radix-4.

2 Related Work.

Claesen et al. [VCM94] and Leeser and O'Leary [LO95] have used theorem provers to verify a non-restoring divider and a radix-2 subtractive square root algorithm, respectively. The circuits verified in both of these efforts are not based on the SRT method and hence do not contain the kinds of optimizations used in SRT division. Recently, German and Clarke [Ger95, CG95] performed a verification of Taylor's SRT divider circuit considered in this paper by manually deriving a set of inequalities that the circuit imposes on the data path signals and then showing, in the MAPLE symbolic algebraic system, that two main SRT correctness invariants are preserved by the data path inequalities. This work provided the main impetus for our work. Clarke et al. [CGZ96] have independently mechanized their verification in the ANALYTICA theorem prover. Our work not only mechanizes all the steps in the verification of the SRT circuit, but also formalizes the general SRT theory correctness and develops a modular framework which can be used to verify other similar circuits. While their specification interprets signals in the circuit as arbitrary real numbers, we interpret signals as parameterized finite, but arbitrary-length bit-vectors.

Methods based on ordered BDDs and symbolic model checking are not well-suited for verifying multipliers and dividers since BDD graphs for such operations grow exponentially with the word size [Bry94]. However, Bryant [Bry95] has used BDDs to check the relation that one iteration of the SRT circuit must preserve for the circuit to correctly divide. To do the verification, he needed to construct a gate-level representation of a *checker-circuit* (much larger than the verified circuit) to describe the desired behavior of the verified circuit, which is not the ideal level of specification.

While Bryant's BMDs can be used to verify multipliers against their number-theoretic specification [Bry94], they cannot be used for SRT verification, because they cannot efficiently check inequalities over bit-vectors. But Clarke and Zhao [CZ95] have recently extended the symbolic model-checking algorithm used in SMV to express and verify word-level properties on numbers. They use an extension of BDDs called *hybrid decision diagrams* to represent integer functions and check relations on them. The word-level model-checker can be used to check if finite-sized arithmetic circuits satisfy desired number-theoretic properties. They have used the word-level model checker to verify Taylor's SRT circuit by checking if a state transition model of the circuit satisfied the main SRT invariants. Both [CZ95] and [CGZ96] are only applicable for fixed-sized data paths.

3 An Overview of PVS

The PVS system combines an expressive specification language with a productive, interactive proof checker that has a reasonable amount of theorem proving capabilities, and has been used for reasoning in domains as diverse as microprocessor verification, protocol verification, and algorithm and architectures concerning fault-tolerance [ORSvH95]. The PVS specification language builds on classical typed higher-order logic with the usual base types, function type constructor, dependent types, and abstract data types. A distinctive feature of PVS are *predicate subtypes* {`x:A | P(x)`}. These subtypes consist of exactly those elements `a` of type `A` satisfying predicate `P(a)`. Predicate subtypes are used to explicitly constrain the domain and ranges of operations in a specification and to define partial functions. In general, type-checking with predicate subtypes is undecidable, and the type-checker generates *type correctness conditions* (TCCs) corresponding to predicate subtypes.

Proofs in PVS are presented in a sequent calculus. The atomic commands of the PVS prover component include induction, quantifier instantiation, automatic conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision diagrams. PVS has an LCF-like strategy language for combining inference steps into more complicated proof strategies.

4 SRT Division

SRT dividers [McS61, Rob58, Toc58] speed up nonrestoring division and are widely used in high-speed floating point units. The quotient is represented in radix-r form and one digit of it is calculated in each iteration. To obtain fast algorithms, SRT division represents quotient digits using a *redundant digit set* $[-a, \ldots, a]$ so that there can by multiple choices for the most significant quotient digit for a given partial remainder and divisor. The redundancy can be used to correct small errors in one iteration in subsequent iterations. It also allows the quotient digit to be computed in parallel with the partial remainder using an approximation of the partial remainder. The common choices $r = 4$ and $a = 2$ lead to adequate and efficient circuits, since multiplication by 0, 1, and 2 is easy.

The presentation of fundamental concepts about SRT division covers the basic recurrence, the conditions under which the computation converges to a reasonable result, and the quotient selection criterion both in exact and approximate forms. These general arithmetic facts about SRT division are presented in terms of their PVS formalization and are parameterized with respect to algorithm-specific details such as the radix and the set of quotient digits. In Sections 5 and 6 we instantiate these arithmetic facts to verify the correctness of a specific high-speed radix-4 SRT circuit on the bit-level. Note also, that we restrict ourselves in this paper to the verification of fixed-point division kernels of IEEE compliant floating-point division.

Subtractive Division Algorithms. Given two normalized fractions p and d of the form $1.xx \ldots xx_2$, the digit recurrence

$$\begin{aligned} p_0 &= p \\ p_{i+1} &= r * (p_i - q_i * d) \text{ with the constraint } |p_{i+1}/d| \leq r * \rho \\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{where } \rho = r * a/(r-1) \end{aligned}$$

computes the value of p/d by producing one quotient digit q_i and a new partial remainder p_{i+1} in each iteration i. The constraint on the partial remainder is needed to guarantee convergence of the algorithm. The above characteristics of subtractive division algorithms are formalized in $\boxed{1}$ for arbitrary radices `r:upfrom[2]` and sets of quotient digits `subrange[-a,a]` such that `a:posnat` and `r/2 <= a < r - 1`.

$\boxed{1}$

```
p_new, p: VAR rational; q: VAR subrange[-a, a]; d: VAR posrat

recurrence?(p_new, p, q, d): bool = (p_new = r * (p - q * d))
rho: rational = a / (r - 1)
p_over_d_bound?(d, p): bool = (-r * rho <= p / d & p / d  <= r * rho)
```

Convergence. The function `valq(i + 1, q)` in $\boxed{2}$ computes the radix-`r` fixed-point value of the accumulated quotient digits `q(0).q(1) ...q(i)`, and Theorem `convergence` states that `q(0).q(1)...q(i)` is an approximation to the infinite precision fraction `p(0) / d` within an error bound.

```
                                                                       2
i, j, k: VAR nat;                d: VAR posrat
p      : VAR sequence[rational]; q: VAR sequence[subrange[-a, a]]

val(i, q): RECURSIVE rational =
 IF i = 0 THEN 0 ELSE  q(i - 1) * 1/r^(i - 1) + val(i - 1, q) ENDIF
MEASURE i

lemma1: LEMMA
 (FORALL j: recurrence?(p(j + 1), p(j), q(j), d))
   IMPLIES p(0) / d - val(i, q) = 1/r^i * (p(i) / d)

convergence: THEOREM
 ((FORALL j: recurrence?(p(j + 1), p(j), q(j), d)) AND
  (FORALL k: p_over_d_bound?(d, p(k))))
   IMPLIES LET residue = p(0) / d - val(i + 1, q) IN
            -1/r^i * rho <= residue & residue <= 1/r^i * rho
```

This theorem is an immediate consequence of the invariant `lemma1` and the given bound on `p(i + 1) / d`, and `lemma1` is proven automatically in PVS with the general-purpose induction strategy `induct-and-simplify` and some basic facts from the library about rational numbers.

Quotient Selection. The hard part in each iteration is to determine a quotient digit `q(i)` such that the next partial remainder `p(i + 1)` also satisfies the boundary constraint `p_over_d_bound?`. By substituting the recurrence relation defining the new partial remainder into the bound contraint on the partial remainder, one can obtain the condition `legitimate?` that characterizes a selection interval of legitimate choices of quotient digits.

```
                                                                       3
q: VAR subrange[-a, a]; d: VAR posrat; p: VAR rat

legitimate?(q, d, p): bool = q - rho <= p/d  &  p/d <= q + rho

lemma2: LEMMA recurrence?(p_new, p, q, d) IMPLIES
               (p_over_d_bound?(d, p_new) IFF legitimate?(q, d, p))
```

Note that the boundaries of this interval depend on the divisor `d`, and Figure 1 graphically displays the region for legitimately selecting quotient digits `-2` through `2` for the choices `r = 4` and `a = 2` (thus `rho = 2/3`). The region for legitimately selecting `q = 1`, for example, is bound by the dashed lines `5/3 * d` and `1/3 * d`.

For specific interpretations, say `r = 4` and `a = 2`, the combination of decision procedures with rewriting on known facts about real and rational numbers (`grind :theories "real_props"`) discharges the proof obligation `lemma2` in $\boxed{3}$ automatically. In the general case, however, where `r` and `a` are uninterpreted, the proof of this fact involves solving non-linear inequalities, and the

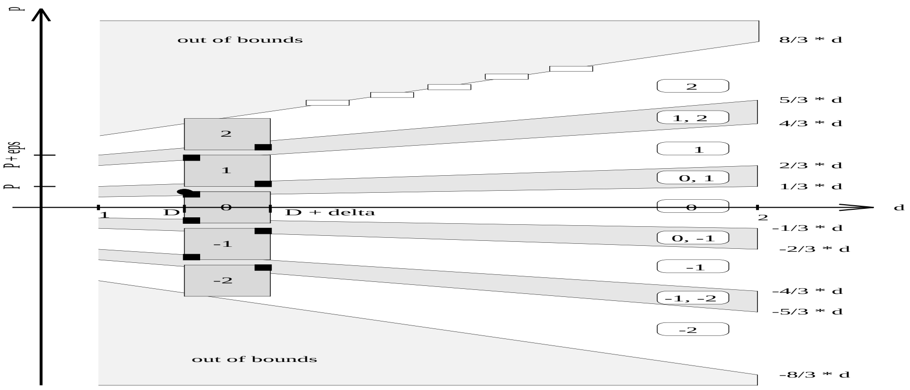

Fig. 1. pd-plot for `r = 4` and `a = 2`

PVS prover needs two interactions to guide the manipulation of these non-linear inequalities.

Redundancy. Shaded regions in Figure 1 indicate pairs `(d, p)` for which selection intervals for the quotient digit `q` overlap. This redundancy permits calculating `q` from truncated versions `P:rational` and `D:posrat` of the partial remainder and divisor respectively.

$\boxed{4}$

```
D: VAR posrat; P: VAR rational

P_bound_by_D?(D, P): bool =
 -eps - r * rho * (D + delta) < P & P < r * rho * (D + delta)

lemma3: LEMMA (P <= p & p < P+eps & D <= d & d < D + delta AND
  p_over_d_bound?(d,p)) IMPLIES P_bound_by_D?(D, P)
```

Let `delta, eps:posrat` be two arbitrary positive rational numbers. Assuming that `P` and `D` underestimate `p` and `d`, respectively, the constraint `p_over_d_bound?` imposed by the algorithm on the partial remainder, imposes a corresponding bound on `P` as a function of `D`. This constraint is defined and proved (by `lemma3`) in $\boxed{4}$. Note that if negative numbers are represented in 2-complement form, which is what we assume in the circuit we verify later, truncation (after the binary point) always produces a number less than the actual value.

Inspection of the *pd-plot* in Figure 1 reveals that the legitimacy of quotient selection for the marked corners of the shaded rectangles suffices to show the legitimacy of selecting this quotient digit for all `(d, p)` pairs in the rectangle. Consequently, the constraint `lookup_legitimate?` in $\boxed{5}$ on lookup tables guarantees the legitimacy of quotient selection as shown in `lemma4`. The combination of the PVS decision procedures with facts from the library about rational numbers proves `lemma4` automatically when `r` and `a` are instantiated with specific numeric values; otherwise manual guidance is needed to deal with non-linear equalities and inequalities.

5

```
lookup_legitimate?(q, D, (P: rational | P_bound_by_D?(D, P))): bool =
 COND
  q = a     -> (a - rho) * (D + delta) <= P,
  0<q & q<a -> (q - rho) * (D + delta) <= P & (P + eps) <= (q+rho)*D,
  q = 0     -> -rho * D <= P & (P + eps) <= rho * D,
 -a<q & q<0 -> (q - rho) * D <= P & (P + eps) <= (q + rho) *(D+delta),
  q = -a    -> (P + eps) <= (-a + rho) * (D + delta)
 ENDCOND

 lemma4: LEMMA
  (P <= p & p < P + eps  AND D <= d & d < D + delta AND
   p_over_d_bound?(d, p) AND lookup_legitimate?(q, D, P))
    IMPLIES legitimate?(q, d, p)
```

Quotient Prediction. A significant reduction of the overall cycle time is obtained by computing the next partial remainder `p(i + 1)` and predicting a next quotient digit `q(i + 1)` in parallel. In this case, the approximation `P(i)` used in iteration i, (under)estimates the next partial remainder `p(i + 1)`. Note that `P(i)` can be computed faster than `p(i + 1)`, since most of the time taken to compute `p(i + 1)` is a full-precision addition, and the computation of `P(i)` only involves a limited-precision adder.

It is a simple matter of combining the results in $\boxed{3}$,$\boxed{4}$,$\boxed{5}$ to prove the statement `invariant` in $\boxed{6}$ for SRT dividers with quotient prediction where the non-trivial part of the induction step involves the chain of implications

```
   legitimate?(q(i), d, p(i))
⇒ remainder_bound?(d, p(i + 1))
⇒ estimation_bound?(D, P(i))
⇒ lookup_legitimate?(q(D, P(i)), D, P(i))
⇒ legitimate?(q(i + 1), d, p(i + 1))
```

6

```
invariant: THEOREM
 (p_over_d_bound?(d, p(0))) AND (legitimate?(q(0), d, p(0))) AND
  (FORALL j: recurrence?(p(j + 1), p(j), q(j), d) AND
    P(j) <= p(j+1) & p(j+1) < P(j)+eps & D <= d & d < D+delta AND
     (P_bound_by_D?(D,P(j)) IMPLIES lookup_legitimate?(q(j+1),D,P(j))))
  IMPLIES
   (p_over_d_bound?(d, p(i)) AND legitimate?(q(i), d, p(i)))
```

Altogether, to prove the correctness of a specific SRT divider circuit it suffices to show that *1)* the arithmetic interpretations of the computed sequences of partial remainders and quotient digits satisfy the recurrence relation `recurrence?`,

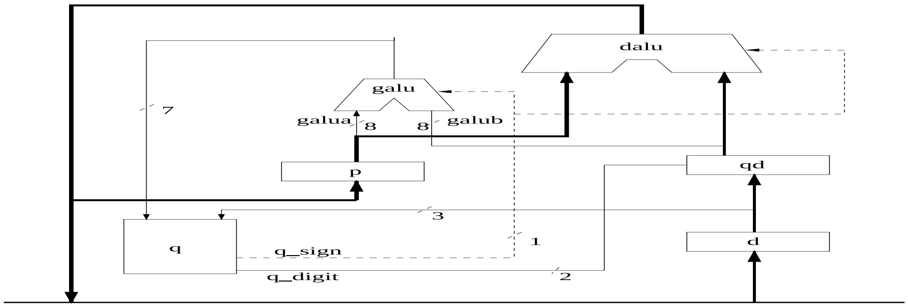

Fig. 2. The data path for the division circuit.

2) there are constants `delta` and `eps` such that the divisor and the partial remainders are bound by under-estimators in the sense described above, and *3)* the quotient selection logic satisfies the `lookup_legitimate?` predicate. Whenever these conditions hold, theorem `invariant` in $\boxed{6}$, and consequently theorem `convergence` in $\boxed{2}$, is applicable.

5 Modeling The Data Path

Now, the data path of an SRT division circuit with `r = 4` and `a = 2` as described by Taylor [Tay81] is specified and proven to be correct by applying the general SRT theory developed in Section 4.

The signals of the circuit in Figure 2 are declared as uninterpreted constants of signals of bit-vectors of various fixed lengths, and the uninterpreted constant `N:posnat`, where `N > 8`, determines the width of the data paths for the divisor and the partial remainders; examples of signal declarations and their interpretation functions are listed in $\boxed{7}$.

```
                                                                    7
d: signal[bvec[N]]; d(i): rational = fp[1,N-1].val(d(i))
P: signal[bvec[7] ; P(i): rational = fp2c[4, 3].val(P(i))
```

The divisor signal `d` has a fixed-point interpretation with `1` leading and `N-1` residual bits, and the estimation `P` of the next partial remainder has a 2-complement fixed-point interpretation with `4` leading bits and `3` residual bits. Note also that overloading the name of the bit-vector signal with its arithmetic interpretation mimics a specification style often found in textbooks about computer arithmetic.

The inputs to the quotient selection unit `q` are the three bit truncation of the divisor `d` and the seven bit approximation `P` of the next partial remainder.

```
                                                                    8
lookup((D: bvec[3]),
       (P: bvec[7] | P_bound_by_D?(1+fp[0,3].val(D),fp2c[4,3].val(P))))
:{ q: subrange(-2, 2) |
    lookup_legitimate?(q, 1 + fp[0,3].val(D), fp2c[4, 3].val(P)) }
```

Here, predicate subtypes serve as a specification of a set of quotient look-up tables by means of domain and range constraints, and a specific implementation of these constraints is proven correct in Section 6. The behavior of the circuit is specified by equality and inequality (to capture the effect of truncation) constraints on the inputs and outputs of the `dalu` and `galu` components which are omitted here for lack of space.

From these formalizations, the `grind` strategy proves the lemmas in 9 about Taylor's division circuit in Figure 2.

```
                                                                        9
taylor_lemma1: LEMMA recurrence?(p(t + 1), p(t), q(t), d(t))
taylor_lemma2: LEMMA galu(t) <= dalu(t) & dalu(t) < galu(t) + 2 *ulp(6)
taylor_lemma3: LEMMA P(t) <= p(t + 1) & p(t + 1) < P(t) + 3/16
```

Together with the constraint on `D` with respect to `d`, this accomplishes Step 2 with `delta = 1/8` and `eps = 3/16` mentioned in Section 4. Now it is a simple matter of instantiating the theorem `invariant` in 6 and `convergence` in 2 to obtain the invariant results in 10 for this specific circuit.

```
                                                                        10
taylor_invariant: LEMMA
 p_over_d_bound?(d(0), p(i)) AND legitimate?(q(i), d(0), p(i))

taylor_convergence: THEOREM
 LET residue =  p(0) / d(0) - val(i + 1, q) IN
  - 2 / (3 * 4^i) <= residue & residue <= 2 / (3 * 4^i)
```

6 The Look-Up Table.

The legitimacy constraint `lookup_legitimate?` (see 8) on quotient look-up tables permits different implementations, and Taylor [Tay81] develops a particularly compact one. This table computes the next quotient digit from the truncation `D:bvec[3]` of the divisor to the three leading bits and the estimation `P:bvec[7]` of the next partial remainder. Bits 6 down to 2 of `P` are used as a table index and the remaining bits are used in some cases to compute the resulting value.

The formalization of the resulting table `q(D, P)` (shown in Appendix A) uses the `TABLE` construct of the PVS specification language [ORS95]. This construct was added to the PVS specification language in order to provide visually appealing two-dimensional tabular specifications in the manner advocated by Parnas and others [Par95]. It proved adequate to express the look-up table of this SRT circuit in a concise and perspicuous way. In particular, blank entries in the look-up table cause the type-checker to generate TCCs which ensure that viable arguments `D`, `P` never point to such a blank entry. Furthermore, the table construct requires that the look-up is functional and ensures this by generating *disjointness* and *coverage* TCCs. From the fact

```
                                                                    11
(FORALL (D, (P: bvec[7] | estimation_bound?(valD(D), valP(P)))):
  lookup_legitimate?(q(D, P), valD(D), valP(P)))
```

one concludes that the given table `q` indeed satisfies the constraint given for look-up tables in Section 5. A simple case split on the different values of `D` and `P` followed by unfolding definitions, term rewriting, and calls to the decision procedures proves the theorem in $\boxed{11}$. The type correctness conditions generated by the type-checker for the look-up table are proven with similar strategies.

In the course of proving the consistency of the look-up table, PVS has proven helpful as a debugging tool and came up with precise *counterexamples*.[2] By injecting, for example, a wrong value `0` at a certain position in the look-up table and rerunning the proof above, the PVS prover returns an unsolved subgoal that yields an immediate counterexample. Note that the 5 missing entries in the look-up table of initial releases of the Pentium floating-point unit were also right at the upper boundary of the legitimate selection region for `q = 2` as depicted in Figure 1 by the blank rectangles.

7 Summary and Conclusions

We have shown how PVS can be used to specify and prove correctness of a non-trivial SRT division algorithm and its hardware implementation in a modular way. This modular approach not only structures the specifications and the proof in a nice way but also has the advantage that slight variations of this particular circuit design and look-up table can be verified by just redoing one part of the proof. Moreover, parts of the theory can be reused for verifying other similar division, and perhaps even square-root, circuits.

This verification exercise demonstrates the value of efficient decision procedures and the use of an expressive specification language in mechanized verification. The concepts of predicate subtypes, overloading, and tables of the PVS specification language proved to be very useful for expressing the high-level designs of this arithmetic circuit in a concise and natural way. Such high-level descriptions reduce the possibility of introducing errors in initial design specification and can also serve as design documents. The tight integration of decision procedures with rewriting strategies of PVS proved to be a useful workhorse, since the circuit specific theorems and the correctness of the table implementation are proven in a fairly automatic way. In most proof obligations that involve non-linear equalities, however, the PVS prover must be manually guided to construct the proofs.

This case study also suggests some improvements to the implementation of PVS. The correctness proof of the table implementation in Section 6 takes 3

[2] Even though the original design of Taylor's look-up table in [Tay81] proved to be correct, we still managed to accidentally inject errors in the initial PVS transcriptions.

hours. This is unreasonably slow, since the proof basically involves small case analysis followed by the evaluation of ground predicates. The incorporation of an efficient notion of evaluation into the proving process could drastically reduce the time for doing this and many other hardware-related proofs. In the future we plan to extend this case study to the verification of related circuits and operations, such as square root, and investigate other concepts like IEEE compliant rounding [Min95].

References

[Atk68] D.E. Atkins. Higher-radix Division Using Estimates of the Divisor and Partial Remainders. *IEEE Transactions on Computers*, C-17(10):925–934, October 1968.

[Bry94] R.E. Bryant. Verification of Arithmetic Functions with Binary Moment Diagrams. Technical Report CMU-CS-94-160, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1994.

[Bry95] R.E. Bryant. Bit-Level Analysis of an SRT Divider Circuit. Technical Report CMU-CS-95-140, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, April 1995.

[CG95] E.M. Clarke and S.M. German. Personal Communication, 1995.

[CGZ96] E.M. Clarke, S.M. German, and X. Zhao. Verifying the SRT Division Algorithm using Theorem Proving Techniques. Submitted to CAV'96, 1996.

[CZ95] E.M. Clarke and X. Zhao. Word Level Symbolic Model Checking: A New approach for Verifying Arithmetic Circuits. Technical Report CMU-CS-95-161, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, April 1995.

[Ger95] S.M. German. Towards Automatic Verification of Arithmetic Hardware. Lecture notes, March 1995.

[LO95] M. Leeser and J. O'Leary. Verification of a Subtractive Radix-2 Square Root Algorithm and Implementation. In *Proc. of ICCD'95*, pages 526–531. IEEE Computer Society Press, 1995.

[McS61] O.L. McSorley. High-speed Arithmetic in Binary Computers. In *Proc. of IRE*, pages 67–91, 1961.

[Min95] P.S. Miner. Defining the IEEE-854 floating-point standard in PVS. NASA Technical Memorandum 110167, NASA Langley Research Center, Hampton, VA, June 1995.

[OF94] S.F. Oberman and M.J. Flynn. Design Issues in Floating-Point Division. Technical Report CSL-TR-94-647, Dept. of Computer Science, Stanford University, Stanford, CA 94305-2140, December 1994.

[ORS95] S. Owre, J. Rushby, and N. Shankar. Analyzing Tabular and State-Transition Specification in PVS. Technical Report CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA, June 1995.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Par95] D. L. Parnas. Using mathematical models in the inspection of critical softwa re. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Ap-*

plications of Formal Methods, International Series in Computer Science, chapter 2, pages 17–31. Prentice Hall, 1995.

[Pra95] V. Pratt. Anatomy of the Pentium Bug. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, pages 97–107. Springer Verlag, May 1995.

[Rob58] J.E. Robertson. A new Class of Digital Division Methods. In *IRE Trans. on Electron. Computers*, volume EC-7, pages 218–222, 1958.

[Tay81] G.S. Taylor. Compatible Hardware For Division and Square Root. In *Proceedings of the 5th Symposium on Computer Arithmetic*, pages 127–134. IEEE Computer Society Press, 1981.

[Toc58] K.D. Tochter. Techniques of Multiplication and Division for Automatic Bin ary Computers. In *Quart. J. Mech. Appl. Match*, volume Part 3, pages 364–384, 1958.

[VCM94] D. Verkest, L. Claesen, and H. De Man. A Proof of the Nonrestoring Division Algorithm and its Implementation on an ALU. *Formal Methods in System Design*, 3:5–31, January 1994.

A Implementation of the Lookup Table

```
q(D, (P | P_bound_by_D?(...,...))): subrange(-2,2) =
 LET a= -(2-P(1)*P(0)), b= -(2-P(1)), c= 1+P(1), d= -(1-P(1)), e= P(1)
 IN TABLE bv2pattern(P^(6,2), bv2pattern(D)
                   |[ 000| 001| 010| 011| 100| 101| 110| 111]|
              %-------------------------------------------%
              |01010|    |    |    |    |    |    |    | 2  ||
              |01001|    |    |    |    |    | 2  | 2  | 2  ||
              |01000|    |    |    |    | 2  | 2  | 2  | 2  ||
              |00111|    |    | 2  | 2  | 2  | 2  | 2  | 2  ||
              |00110|    | 2  | 2  | 2  | 2  | 2  | 2  | 2  ||
              |00101| 2  | 2  | 2  | 2  | 2  | 2  | 2  | 1  ||
              |00100| 2  | 2  | 2  | 2  | c  | 1  | 1  | 1  ||
              |00011| 2  | c  | 1  | 1  | 1  | 1  | 1  | 1  ||
              |00010| 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  ||
              |00001| 1  | 1  | 1  | 1  | e  | 0  | 0  | 0  ||
              |00000| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  ||
              |11111| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  ||
              |11110| -1 | -1 | d  | d  | 0  | 0  | 0  | 0  ||
              |11101| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 ||
              |11100| a  | b  | -1 | -1 | -1 | -1 | -1 | -1 ||
              |11011| -2 | -2 | -2 | b  | -1 | -1 | -1 | -1 ||
              |11010| -2 | -2 | -2 | -2 | -2 | -2 | b  | -1 ||
              |11001| -2 | -2 | -2 | -2 | -2 | -2 | -2 | -2 ||
              |11000|    |    | -2 | -2 | -2 | -2 | -2 | -2 ||
              |10111|    |    |    | -2 | -2 | -2 | -2 | -2 ||
              |10110|    |    |    |    |    | -2 | -2 | -2 ||
              |10101|    |    |    |    |    |    | -2 | -2 ||
              %-------------------------------------------%
```